AUUG Security Symposium
19-21 November 2001

Brisbane

TUTORIAL M5

Building an Open Source Firewall,
by Michael Paddon

http://www.auug.org.au/security2001/

AUUG Inc
PO Box 366, Kensington NSW 2033 Australia

Phone: 1-800 625 655 or +61-2-8824 9511
Fax: +61-2-8824 9522
Email: auug@auug.org.au

http://www.auug.org.au

UNIX® AND OPEN SYSTEMS USERS

Building
An Open Source Firewall

Michael Paddon

michael@paddon.org
http://www.paddon.org/mwp

Darren Reed's IPFilter

Features:

- Packet filtering
- Stateful flows
- Fragment handling
- Network address translation
- Redirection
- Transparent proxying
- Traffic logging
- Packet authentication

- Open Source

Operating Systems

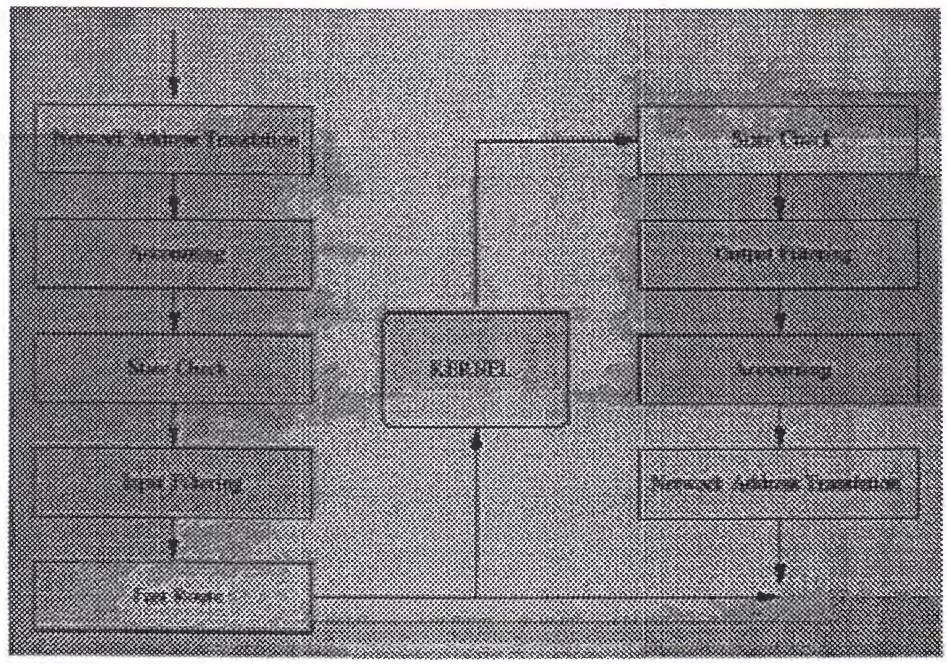
Ships with:

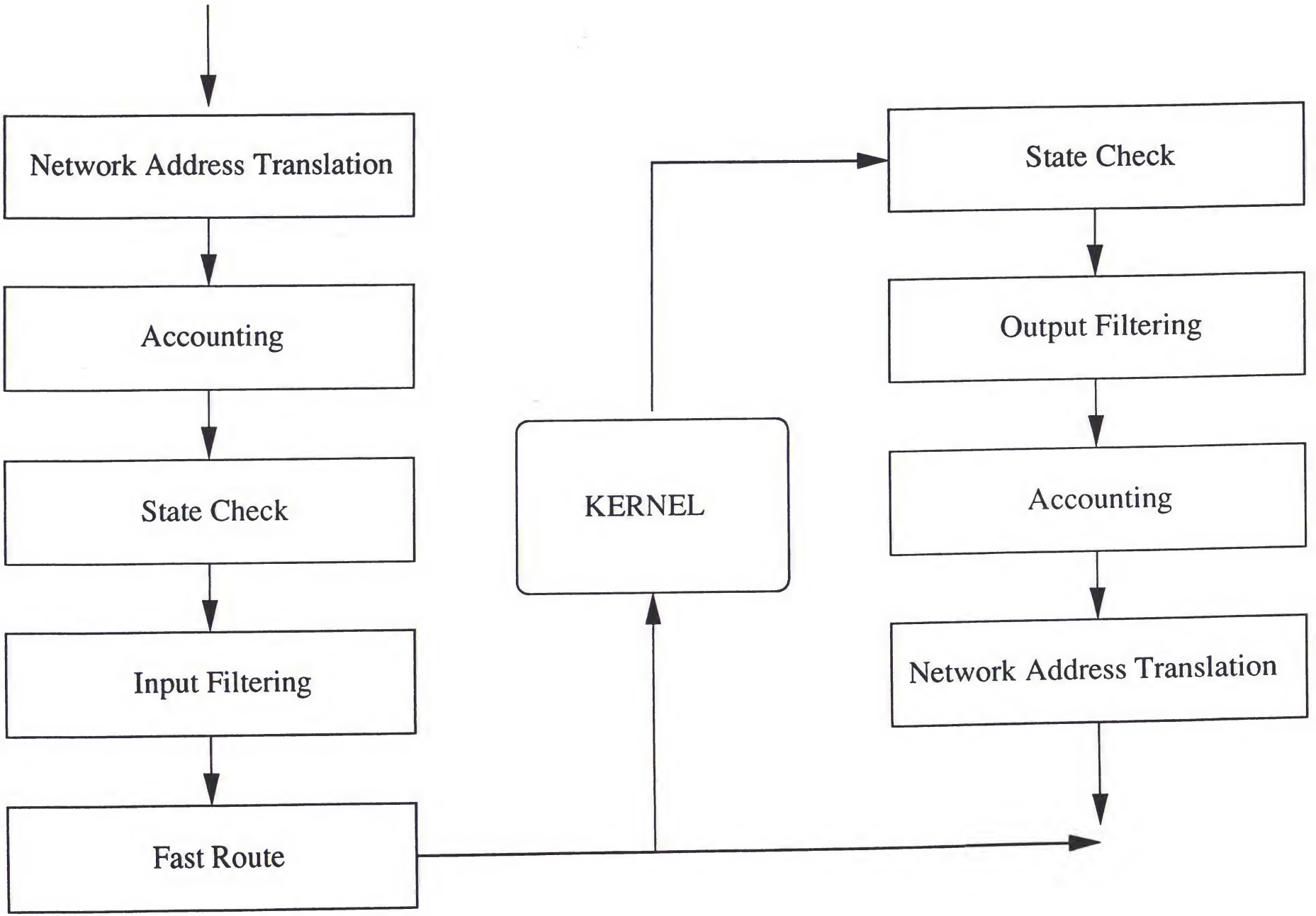
- FreeBSD (post 2.2)
- NetBSD (post 1.2)
- OpenBSD (2.0 to 2.9)

Supported on:

- BSD/OS
- HP/UX
- Irix
- QNX
- Solaris
- SunOS

Processing Model




Network Address Translation
Accounting
State Check
Input Filtering
Fast Route
KERNEL
State Check
Output Filtering
Accounting
Network Address Translation

Rule Sets

IPFilter is configured by a ruleset:

- A ruleset consists of a sequence of rules, against which each packet is matched in turn.
- The last matching rule determines the packet's fate. If no rule matches, a default action is applied.
- All rules are specified as either "in" or "out", depending on whether they apply to arriving or departing packets.
- Comments are delimited by '#' and newline.

Basic Rules

The basic format of a rule is:

[insert] action **in | out** *[options] [tos] [ttl] [proto] [ip] [group]*

The two most common actions are:

- "pass": allow a packet to be transmitted
- "block": prevent a packet from being transmitted

An example (small) ruleset:

```
block in all
block out all
pass in from any to 10.0.0.1
pass out from 10.0.0.0/8 to any
```

Default Actions

- The default action for unmatched packets is defined at compile time, and is either "pass" or "block".
- Depending on a specific default behaviour is dangerous.
- Consider starting all rulesets with:

```
block in all
block out all
```

Sources and Destinations

Packets may be constrained by source and destination with the syntax:

from [**!**]*object* **to** [**!**]*object*

Objects may be specified as:

- "any"
- a DNS name or a dotted quad address, optionally followed by:
  - /N
  - mask N.N.N.N
  - mask 0xNNNNNNNN

The string "all" is shorthand for "from any to any".

Source and Destination Examples

Host Specification:

```
pass out from 10.0.0.1 to any
pass out from 10.0.0.1/32 to any
pass out from 10.0.0.1 mask 255.255.255.255
    to any
```

Network Specification:

```
pass in from any to 192.168.0.0/24
pass in from any
    to 192.168.0.0 mask 255.255.255.0
pass in from any
    to 192.168.0.0 mask 0xffffff00
```

There is no support (yet) for IPv6 addresses.

Object Negation

Sources and destinations may be negated by a preceding "!".

This is useful for filtering everything but a given host or network.

Examples:

```
pass in from !192.168.0.0/24 to any
pass in from any to !10.1.0.0/16
```

Use of Hostnames Considered Harmful

- What if the DNS is down when you configure?
- What if your DNS is subverted?
- What if your sysadmin changes a DNS record without checking with you?
- What if your ruleset depends on hostnames hosted on third party DNS servers?

Recommendation:

- For better security and reliability, only use numeric addresses in your ruleset.

Filtering by Protocol

A protocol may be specified with the "proto" keyword, preceding the source and destination specification.

The following protocols are recognised:

- "tcp/udp", "udp", "tcp", "icmp"
- a decimal number

For example:

```
pass in proto tcp from any to any
pass in proto tcp/udp from 192.168.0.0/24
    to 192.168.1.0/24
pass in proto 50 from any to 10.0.0.1
```

TCP and UDP Port Filtering

When filtering TCP and UDP packets, you can specify a port constraint following a source or destination address.

The general syntax is:

port *comparison port-id*

Supported comparisons:

- "=", "eq"
- "!=", "ne"
- "<", "lt"
- ">", "gt"
- "<=", "le"
- ">=", "ge"

Ranged Port Filtering

Port range constraints are also supported, with the syntax:

port *port-id range port-id*

Supported range operators:

- "<>" (inclusive range)
- "><" (exclusive range)

This syntax is generally more convenient to use when manipulating a "window" of ports.

Ports may be specified as a service name or a decimal number.

Port Filtering Examples

Comparisons:

```
pass in proto tcp from any
    to 10.0.0.1 port = ssh
pass in proto udp from any
    to 10.0.0.1 port = domain
pass in proto tcp from any
    to 10.0.0.1 port >= 49152
```

Ranges:

```
pass in proto udp from any
    to 10.0.0.1 port 7000 <> 7009
pass in proto udp from any
    to 10.0.0.1 port 6999 >< 7010
```

TCP Flags

When filtering TCP packets, you can specify TCP flag constraints, following the source and destination specification.

The general syntax is:

flags *wanted-flags*[/*tested-flags*]

Supported flags:

- "F": FIN
- "S": SYN
- "R": RST
- "P": PUSH
- "A": ACK
- "U": URG

TCP Flag Examples

"Unidirectional" TCP:

```
pass in proto tcp from 10.0.0.1 to any
pass in proto tcp from any to 10.0.0.1
block in proto tcp from any
    to 10.0.0.1 flags S/SA
```

A more stringent approach:

```
pass in proto tcp from 10.0.0.1 to any
pass in proto tcp from any to 10.0.0.1
block in proto tcp from any
    to 10.0.0.1 flags A/A
```

Miscellaneous IP Constraints

Various other IP constraints may be specified, following the TCP flags specification.

The general syntax is:

with [not | no] ipopts | short | frag | opt *option-list ...*

Supported options:

- "nop", "rr", "lsrr", "ssrr", "mtup", "mtur"
- "ts", "tr", "encode", "sec", "e-sec", "cipso"
- "satid", "addext", "visa", "imitd", "eip", "finn", "zsu"

More than one option may be specified, separated by commas.

Security Options

The RFC 1038 security option supports additional discrimination. These are specified alongside other options, with the syntax:

sec-class *level*

Supported security levels:

- "unclass", "confid"
- "reserv-1", "reserv-2"
- "reserv-3", "reserv-4"
- "secret", "topsecret"

Miscellaneous Constraint Examples

Blocking various classes of packet:

```
block in all with no ipopts
block in all with no short
block in all with no frag
```

Passing specific options:

```
pass in all with opt tr
pass in from 10.0.0.1 to 10.0.0.2
    with opt sec-class unclass
```

Filtering ICMP

When filtering ICMP packets, ICMP type and code constraints may be specified, following the source and destination specification.

The general syntax is:

icmp-type *icmp-type* **code** *decimal-number*

Supported ICMP types:

- "unreach", "echo", "echorep", "squench", "redir"
- "timex", "paramprob", "timest", "timestrep"
- "inforeq", "inforep", "maskreq", "maskrep"
- a decimal number

Filtering ICMP Examples

Permitting useful ICMP:

```
block in proto icmp

pass in proto icmp from any to any
    icmp-type echo
pass in proto icmp from any to any
    icmp-type echorep

pass in proto icmp from any to any
    icmp-type timex
```

Any ICMP can be theoretically dangerous, but watch our especially for "unreach", "squench", "redir" and "paramprob".

Stateful Filtering

IPFilter can track TCP, UDP and ICMP flows. This is achieved by requesting "keep state", following source and destination specification.

For example:

```
pass in proto tcp from 10.0.0.0/8
    to any flags S/SAFR keep state
pass in proto udp from 10.0.0.0/8
    to any port = domain keep state
pass in proto udp from 10.0.0.0/8
    to any port = ntp keep state
pass in proto icmp from 10.0.0.0/8
    to any icmp-type echo keep state
```

TCP and UDP State Tracking

TCP state is tracked by looking for packets returning to the initiating address and port in accordance with the TCP state machine, with the following typical timeouts:

- Idle connection: 5 days
- Half closed connection: 2 hours
- Partially closed connection: 4 minutes
- Closed connection: 2 minutes

UDP flows are tracked by looking for packets returning to the initiating address and port, with the following typical timeouts:

- Waiting for response: 4 minutes
- Received response: 24 seconds

ICMP State Tracking

ICMP state is tracked by looking for corresponding ICMP return packets to the initaiting address, with the following typical timeouts:

- Waiting for response: 2 minutes
- Received response: 12 seconds

Corresponding ICMP packets are defined as:

- echo -> echorep
- timest -> timestrep
- inforeq -> inforeq
- maskreq -> maskrep

Fragment State Tracking

IPFilter can also track fragment flows. This is achieved by requesting "keep frags", following the source and destination specification.

For example:

```
pass in from 10.0.0.0/24
    to 10.0.1.0/24 keep frags
block in from any
    to any with frag
```

Time to Live

The time to live (TTL) value in the IP header may be specified preceding the protocol clause.

The general syntax is:

ttl *decimal-number*

Only an exact TTL may be matched.

For example:

```
pass in ttl 64 from any to any
```

Type of Service

The type of service (TOS) field in the IP header may be specified preceding the TTL clause.

The general syntax is:

ttl *decimal-number* | *hex-number*

Only an exact TOS may be matched.

For example:

```
# pass low delay packets
pass in tos 0x10 from any to any
```

Generating Return ICMP's

A block rule may generate a return ICMP packet with the syntax:

block [return-icmp(*code*) | return-rst]

Supported ICMP return codes:

- "net-unr", "host-unr", "proto-unr", "port-unr"
- "needfrag", "srcfail", "net-unk", "host-unk"
- "isolate", "net-prohib", "host-prohib", "net-tos"
- "host-tos", "filter-prohib", "host-preced",
- "cutoff-preced"
- decimal number

Return ICMP Examples

Typical initial block rules:

```
block return-icmp(filter-prohib) all
block return-rst proto tcp all
```

Instead of "return-icmp", you may specify "return-icmp-as-dest" to cause the returned packet to appear to originate from the rejected destination.

```
block return-icmp-as-dest(filter-prohib) all
```

Return ICMP and RST functionality only works for "in" rules.

Recommendation: Try to design your rulesets around incoming, not outgoing, traffic.

Packet Logging

Packets may be logged with a "log" action.

The general syntax is:

log [body] [first] [or-block] [level *facility*[**.***priority*]]

Logging Options

- "body": log first 128 bytes of packet.
- "first": log only trigger packets when "keep" is enabled.
- "or-block": if logging fails, block packet.
- "level": when logging via syslog, use specified parameters.

Syslog Levels

Supported facilities:

- "kern", "user", "mail", "daemon", "auth"
- "syslog", "lpr", "news", "uucp", "cron"
- "ftp", "authpriv", "audit", "logalert"
- "local0", "local1", "local2", "local3"
- "local4", "local5", "local6", "local7"

Supported priorities:

- "emerg", "alert", "crit", "err"
- "warn", "notice", "info", "debug"

Default Syslog Level

The default facility is "local0".

The default priority depends on the reason for logging:

- "info": packets logged using the "log" action.
- "notice": packets logged which are also passed.
- "warn": packets logged which are also blocked.
- "err": packets logged which are considered short.

Logging examples

Simple logging actions:

```
log body from 10.0.0.1 to 10.0.0.2
log first proto tcp from 10.0.0.1
    to 10.0.0.2 keep state
log or-block from 10.0.0.1 to 10.0.0.2
log level local1.err from 10.0.0.1 to 10.0.0.2
```

Logging may also be combined with other actions, by following the original action with a log clause:

```
pass log from 10.0.0.1 to 10.0.0.2
block return-rst log first level local1.err
    proto tcp from 10.0.0.1 to 10.0.0.2
    keep state
```

Traffic Accounting

Simple traffic accounting may be accomplished with the "count" action.

The "count" action is special, since it has no effect on whether the packet will be passed or blocked.

For example:

```
count in all
count in from 10.0.0.0/24 to 10.0.1.0/24
```

Traffic statistics may be viewed with ipfstat(8).

Packet Authentication

The "auth" action requests packet authentication by a user space program:

- The packet is buffered and passed, via "/dev/ipauth", to user space.
- The external program performs authentication.
- The result is passed, via "/dev/ipauth", back to IPFilter.
- The packet is handled according to the result value.

For example:

```
auth in from 10.0.0.0/24 to any
```

Pre-Authentication

Rather than process every packet through "auth", temporary rules may be set up (via "ipf -P") which are then applied using the "preauth" action.

For example:

```
preauth in proto tcp
    from 10.0.0.0/24 to any port = ftp
```

If there are no matching rules in the temporary list, the packet is blocked.

Quick Rules

An action may be marked as "quick", which causes ruleset processing
to terminate if the rule matches.

This can be used, among other things, to speed up rulesets.

For example:

```
pass in quick from 10.0.0.1 to any
block in quick proto icmp all with short
```

Interface Rules

Packets may be matched to interfaces by specifying an "on" clause.

This is useful for enforcing topology.

For example:

```
pass in quick on lo0 all
pass out quick on lo0 all

block in quick on de0 from 10.0.0.0/8 to any
block in quick on de0 from 192.168.0.0/16 to any
block in quick on de0 from 172.16.0.0/12 to any
```

Packet Duplication

Packets may be duplicated by specifying a "dup-to" clause.

This is useful for creating logging or sniffing networks.

dup-to *interface*[:*address*|*hostname*]

For example:

```
block in on de0 dup-to de1 proto icmp all
pass in dup-to de1:10.0.0.1
    from any to 192.168.0.0/16
```

Fast Routing

The kernel packet processing may be circumvented by use of a "fastroute" (or "to") clause.

This is useful for building transparent firewalls.

For example:

```
pass in on de1 fastroute de2
    from 10.0.0.0/8 to 192.168.0.0/16
```

Rule Ordering

Rules are normally appended to a ruleset in the order in which they appear.

Prepending a a rule with a "@n" directive causes it to be inserted into the ruleset as the n'th rule.

For example:

```
@10 block in from 10.0.0.254 to any
```

This is useful when dynamically updating rules through ipf(1). Rules may also be removed via this interface.

Rule Skipping

Rules may be skipped by specifying the "skip" action.

This is useful for exempting certain classes of traffic from some rules.

For example:

```
skip 3 in from 10.0.0.0/8 to 192.168.0.0/16
skip 2 in from 192.168.0.0/16 to 10.0.0.0/8
block in proto icmp all
block in proto udp all
```

If a rule is inserted or deleted in a skipped block of rules, the governing skip values are updated automatically.

Rule Groups

Rules may be grouped by appended a group specification.

The general syntax is:

[**head** *decimal-number*] [**group** *decimal-number*]

Rules without a group specification are implicitly in group 0.

Groups are useful for disjoint processing and increasing performance.

Grouping Example

A head rule exists in two groups, the group it was in and the the group it heads.

If a head rule is marked "quick", the entire group is executed before the ruleset terminates.

```
block in quick from any to 10.0.0.0/8 head 10
 block in proto tcp head 20 group 10
   pass in from any to any port = ssh group 20
   pass in from any to any port = imap group 20
 block in proto udp head 30 group 10
   pass in from any to any port = domain group 30
   pass in from any to any port = ntp group 30
```

Effective Ruleset Structures

- Philosophy: everything not explicitly allowed is restricted.
- Use quick rules sparingly.
- Use groups to simplify and increase performance.
- Use quick groups to implement a "switch" semantic.
- Track state to reduce backchannel exposure.

A useful generic structure:

1. block everything
2. block quick absolutely unwanted packets
3. pass wanted packets

A Simple Complete Policy

```
block return-icmp in log from any to any
block return-rst in log proto tcp from any to any
pass out all

block return-icmp in log quick from any to any with short
block return-icmp in log quick from any to any with ipopts

pass in on de0 quick from to any

pass in on lo0 quick  all

block return-icmp in log quick on ppp0 from 127.0.0.0/8 to any
block return-icmp in log quick on ppp0 from 10.0.0.0/8 to any
pass in on ppp0 proto tcp from any to 10.0.0.1 port = http
pass in on ppp0 proto tcp from any to 10.0.0.1 port = smtp
pass in on ppp0 proto tcp from any to 10.0.0.1 port = 993 # simap
pass in on ppp0 proto tcp from any to 10.0.0.0/8 port = ssh
pass out on ppp0 proto icmp from any to any keep state
pass out on ppp0 proto udp from any to any keep state
pass out on ppp0 proto tcp from any to any keep state
```

Network Address Translation

A separate ruleset defines any network address translation required.

Simple NAT functionality is enabled with the use of the "map" directive:

map *interface network* **->** *network*

For example:

```
map de0 10.0.0.0/8 -> 201.2.3.0/24
```

NAT processing occurs at the specified outgoing interface.

Complex Packet Matching

Sometimes it is useful to be able to NAT differently, based on more than the source address.

For the source part of the NAT rule, you can use an expression of the form:

from *address* **to** *address*

For example:

```
map de0 from 10.0.0.0/8 to any
    -> 201.2.3.0/24
map de0 from 192.168.0.0/16 to 201.0.0.0/8
    -> 201.2.3.0/24
map de0 from 192.168.0.0/16 to 202.0.0.0/8
    -> 202.2.3.0/24
```

Port Mapping

When translating TCP and UDP, the port space can be used to increase the available number of mappings.

portmap tcp|udp|tcp/udp *low-port*:*high-port*
portmap tcp|udp|tcp/udp auto

For example:

```
map de0 10.0.0.0/8 -> 202.2.3.0/32
    portmap tcp/udp 1025:65535
map de0 10.0.0.0/8 -> 202.2.3.0/24
    portmap tcp/udp auto
```

Static Mapping

Usually, mapping is done by dynamically searching for an unused address, or <address, port> tuple. Occasionally it is desirable to map using a static algorithm instead.

This may be achieved with the "map-block" directive.

For example:

```
map-block de0 10.0.0.0/8 -> 202.2.3.0/24
```

Static mapping can be less sensitive to accidental or intentional denial of service conditions.

Bi-directional Mapping

Mapping can be performed in both directions with a "bimap" directive.

For example:

```
bimap de0 10.0.0.0/8 -> 202.2.3.0/24
bimap de0 10.0.0.0/8 -> 202.2.3.0/32
    portmap tcp/udp auto
```

This is useful for connecting peer networks.

Redirection

Redirection of packets can be achieved with a "rdr" directive:

rdr *interface network* **port** *port* -> *address* **port** *port* **tcp|udp|tcp/udp**

For example:

```
rdr de0 0.0.0.0/0 port http
    -> 10.0.0.1 port http tcp
```

Redirection processing occurs on incoming packets.

Load Balancing

Multiple destinations may be specified in a "rdr" directive to achieve simple round-robin load balancing.

For example:

```
rdr de0 0.0.0.0/0 port http
    -> 10.0.0.1,10.0.0.2 port http tcp
```

Adding a "round-robin" directive allows multiple rules to work in this way:

```
rdr de0 0.0.0.0/0 port http
    -> 10.0.0.1 port http tcp round-robin
rdr de0 0.0.0.0/0 port http
    -> 10.0.0.2 port http tcp round-robin
```

Transparent Proxying

Transparent proxies may be implemented by redirecting packets to localhost.

For example:

```
rdr de0 0.0.0.0/0 port http
    -> 127.0.0.1 port http tcp
```

The application must perform an ioctl on "/dev/ipnat" to determine the original source address.

Complex Protocols

IPFilter has a number of built in proxies that can be used to deal with more complex protocols.

These are a accessed via a "map" directive, with a proxy clause following the destination address:

proxy port *port tag/protocol*

These proxies work buy installing temporary rules as required by the specific application protocol.

Supported Proxies

Currently there is proxy support for the following protocols:

- ftp/tcp
- raudio/tcp
- rcmd/tcp
- ipsec/udp

For example:

```
map de0 from 192.168.0.0/16 -> 10.0.0.1/32
    proxy port ftp ftp/tcp
map de0 from 192.168.0.0/16 -> 10.0.0.2/32
    proxy port raudio raudio/tcp
```

Utility Programs

- ipf(1): manage filter rulesets
- ipnat(1): manage NAT rulesets
- ipftest(1): test filter rules

- ipfstat (8): retrieve statistics and rulesets
- ipfs(8): save/restore state information
- ipmon(8): monitor for logged packets

Useful Commands

```
ipf -Fa -f /etc/ipf.rules
ipf -I -Fa -f /etc/ipf.rules
ipf -s
ipf -z

ipfstat -i
ipfstat -o
ipfstat -i -a
ipfstat -s

ipnat -C -f /etc/ipnat.rules
ipnat -F
ipnat -s

ipmon -s -D
```

General Firewall Configuration

- Turn off everything you don't need.
- Don't run non-firewall applications.
- Remove all user accounts.
- Monitor patches, upgrade regularly.
- Disable all remote access.
- Tripwire your box.

- Check your logs!

- Consider auditing your code (or OpenBSD).

References

- IPFilter Home Page (http://www.ipfilter.org/)
- IPFilter HOWTO (http://www.obfuscation.org/ipf/)
- UCD SNMP Home Page (http://ucd-snmp.ucdavis.edu/)

This tutorial was written relative to IPFilter 3.4.21.

The following manual pages are included from the IPFilter 3.4.21 release.
These are governed by the IPFilter licence, reproduced below:

IPFILTER(1)

NAME

IP FIlter

DESCRIPTION

IP Filter is a package providing packet filtering capabilities for a variety of operating systems. On a properly setup system, it can be used to build a firewall.

SEE ALSO

ipf(8), ipf(1), ipf(5), ipnat(1), ipnat(5), mkfilters(1)

NAME

ipftest – test packet filter rules with arbitary input.

SYNOPSIS

ipftest [**–vbdPSTEHX**] [**–I** interface] **–r** <filename> [**–i** <filename>]

DESCRIPTION

ipftest is provided for the purpose of being able to test a set of filter rules without having to put them in place, in operation and proceed to test their effectiveness. The hope is that this minimises disruptions in providing a secure IP environment.

ipftest will parse any standard ruleset for use with **ipf** and apply input, returning output as to the result. However, **ipftest** will return one of three values for packets passed through the filter: pass, block or nomatch. This is intended to give the operator a better idea of what is happening with packets passing through their filter ruleset.

When used without either of **–S**, **–T** or **–E**, **ipftest** uses its own text input format to generate "fake" IP packets. The format used is as follows:

```
"in"|"out" "on" if ["tcp"|"udp"|"icmp"]
        srchost[,srcport] dsthost[,destport] [FSRPAU]
```

This allows for a packet going "in" or "out" of an interface (if) to be generated, being one of the three main protocols (optionally), and if either TCP or UDP, a port parameter is also expected. If TCP is selected, it is possible to (optionally) supply TCP flags at the end. Some examples are:

```
# a UDP packet coming in on le0
in on le0 udp 10.1.1.1,2210 10.2.1.5,23
# an IP packet coming in on le0 from localhost - hmm :)
in on le0 localhost 10.4.12.1
# a TCP packet going out of le0 with the SYN flag set.
out on le0 tcp 10.4.12.1,2245 10.1.1.1,23 S
```

OPTIONS

–v Verbose mode. This provides more information about which parts of rule matching the input packet passes and fails.

–d Turn on filter rule debugging. Currently, this only shows you what caused the rule to not match in the IP header checking (addresses/netmasks, etc).

–b Cause the output to be a brief summary (one-word) of the result of passing the packet through the filter; either "pass", "block" or "nomatch". This is used in the regression testing.

–I <interface>
Set the interface name (used in rule matching) to be the name supplied. This is useful with the **–P, –S, –T** and **–E** options, where it is not otherwise possible to associate a packet with an interface. Normal "text packets" can override this setting.

–P The input file specified by **–i** is a binary file produced using libpcap (i.e., tcpdump version 3). Packets are read from this file as being input (for rule purposes). An interface maybe specified using **–I.**

–S The input file is to be in "snoop" format (see RFC 1761). Packets are read from this file and used as input from any interface. This is perhaps the most useful input type, currently.

–T The input file is to be text output from tcpdump. The text formats which are currently supported are those which result from the following tcpdump option combinations:

```
tcpdump -n
tcpdump -nq
tcpdump -nqt
tcpdump -nqtt
tcpdump -nqte
```

ipftest(1)

-H The input file is to be hex digits, representing the binary makeup of the packet. No length correction is made, if an incorrect length is put in the IP header.

-X The input file is composed of text descriptions of IP packets.

-E The input file is to be text output from etherfind. The text formats which are currently supported are those which result from the following etherfind option combinations:

```
etherfind -n
etherfind -n -t
```

-i <filename>
Specify the filename from which to take input. Default is stdin.

-r <filename>
Specify the filename from which to read filter rules.

SEE ALSO

ipf(5), ipf(8), snoop(1m), tcpdump(8), etherfind(8c)

BUGS

Not all of the input formats are sufficiently capable of introducing a wide enough variety of packets for them to be all useful in testing.

IPNAT(1)

NAME

ipnat – user interface to the NAT

SYNOPSIS

ipnat [**–lnrsvCF**] **–f** <*filename*>

DESCRIPTION

ipnat opens the filename given (treating "–" as stdin) and parses the file for a set of rules which are to be added or removed from the IP NAT.

Each rule processed by **ipnat** is added to the kernels internal lists if there are no parsing problems. Rules are added to the end of the internal lists, matching the order in which they appear when given to **ipnat**.

OPTIONS

–C delete all entries in the current NAT rule listing (NAT rules)

–F delete all active entries in the current NAT translation table (currently active NAT mappings)

–l Show the list of current NAT table entry mappings.

–n This flag (no-change) prevents **ipf** from actually making any ioctl calls or doing anything which would alter the currently running kernel.

–s Retrieve and display NAT statistics

–r Remove matching NAT rules rather than add them to the internal lists

–v Turn verbose mode on. Displays information relating to rule processing and active rules/table entries.

FILES

/dev/ipnat

SEE ALSO

ipnat(5), ipf(8), ipfstat(8)

NAME

mkfilters – generate a minimal firewall ruleset for ipfilter

SYNOPSIS

mkfilters

DESCRIPTION

mkfilters is a perl script that generates a minimal filter rule set for use with **ipfilter** by parsing the output of **ifconfig**.

SEE ALSO

ipf(8), ipf(5), ipfilter(5), ifconfig(8)

NAME

ipf – packet filtering kernel interface

SYNOPSIS

```
#include <netinet/ip_compat.h>
#include <netinet/ip_fil.h>
```

IOCTLS

To add and delete rules to the filter list, three 'basic' ioctls are provided for use. The ioctl's are called as:

```
ioctl(fd, SIOCADDFR, struct frentry **)
ioctl(fd, SIOCDELFR, struct frentry **)
ioctl(fd, SIOCIPFFL, int *)
```

However, the full complement is as follows:

```
ioctl(fd, SIOCADAFR, struct frentry **) (same as SIOCADDFR)
ioctl(fd, SIOCRMAFR, struct frentry **) (same as SIOCDELFR)
ioctl(fd, SIOCADIFR, struct frentry **)
ioctl(fd, SIOCRMIFR, struct frentry **)
ioctl(fd, SIOCINAFR, struct frentry **)
ioctl(fd, SIOCINIFR, struct frentry **)
ioctl(fd, SIOCSETFF, u_int *)
ioctl(fd, SIOGGETFF, u_int *)
ioctl(fd, SIOCGETFS, struct friostat **)
ioctl(fd, SIOCIPFFL, int *)
ioctl(fd, SIOCIPFFB, int *)
ioctl(fd, SIOCSWAPA, u_int *)
ioctl(fd, SIOCFRENB, u_int *)
ioctl(fd, SIOCFRSYN, u_int *)
ioctl(fd, SIOCFRZST, struct friostat **)
ioctl(fd, SIOCZRLST, struct frentry **)
ioctl(fd, SIOCAUTHW, struct frauth_t **)
ioctl(fd, SIOCAUTHR, struct frauth_t **)
ioctl(fd, SIOCATHST, struct fr_authstat **)
```

The variations, SIOCADAFR vs. SIOCADIFR, allow operation on the two lists, active and inactive, respectively. All of these ioctl's are implemented as being routing ioctls and thus the same rules for the various routing ioctls and the file descriptor are employed, mainly being that the fd must be that of the device associated with the module (i.e., /dev/ipl).

The three groups of ioctls above perform adding rules to the end of the list (SIOCAD*), deletion of rules from any place in the list (SIOCRM*) and insertion of a rule into the list (SIOCIN*). The rule place into which it is inserted is stored in the "fr_hits" field, below.

```
typedef struct  frentry {
    struct  frentry *fr_next;
    u_short fr_group;     /* group to which this rule belongs */
    u_short fr_grhead;    /* group # which this rule starts */
    struct  frentry *fr_grp;
    int     fr_ref;       /* reference count - for grouping */
    void    *fr_ifa;
#if BSD >= 199306
    void    *fr_oifa;
#endif
    /*
     * These are only incremented when a packet  matches this rule and
     * it is the last match
     */
```

IPF(4)

```
        U_QUAD_T        fr_hits;
        U_QUAD_T        fr_bytes;
        /*
         * Fields after this may not change whilst in the kernel.
         */
        struct  fr_ip   fr_ip;
        struct  fr_ip   fr_mip; /* mask structure */

        u_char  fr_tcpfm;       /* tcp flags mask */
        u_char  fr_tcpf;        /* tcp flags */

        u_short fr_icmpm;       /* data for ICMP packets (mask) */
        u_short fr_icmp;

        u_char  fr_scmp;        /* data for port comparisons */
        u_char  fr_dcmp;
        u_short fr_dport;
        u_short fr_sport;
        u_short fr_stop;        /* top port for <> and >< */
        u_short fr_dtop;        /* top port for <> and >< */
        u_32_t  fr_flags;       /* per-rule flags && options (see below) */
        u_short fr_skip;        /* # of rules to skip */
        u_short fr_loglevel;    /* syslog log facility + priority */
        int     (*fr_func) __P((int, ip_t *, fr_info_t *));
        char    fr_icode;       /* return ICMP code */
        char    fr_ifname[IFNAMSIZ];
#if BSD > 199306
        char    fr_oifname[IFNAMSIZ];
#endif
        struct  frdest  fr_tif; /* "to" interface */
        struct  frdest  fr_dif; /* duplicate packet interfaces */
} frentry_t;
```

When adding a new rule, all unused fields (in the filter rule) should be initialised to be zero. To insert a rule, at a particular position in the filter list, the number of the rule which it is to be inserted before must be put in the "fr_hits" field (the first rule is number 0).

Flags which are recognised in fr_flags:

```
FR_BLOCK        0x000001   /* do not allow packet to pass */
FR_PASS         0x000002   /* allow packet to pass */
FR_OUTQUE       0x000004   /* outgoing packets */
FR_INQUE        0x000008   /* ingoing packets */
FR_LOG          0x000010   /* Log */
FR_LOGB         0x000011   /* Log-fail */
FR_LOGP         0x000012   /* Log-pass */
FR_LOGBODY      0x000020   /* log the body of packets too */
FR_LOGFIRST     0x000040   /* log only the first packet to match */
FR_RETRST       0x000080   /* return a TCP RST packet if blocked */
FR_RETICMP      0x000100   /* return an ICMP packet if blocked */
FR_FAKEICMP     0x00180    /* Return ICMP unreachable with fake source */
FR_NOMATCH      0x000200   /* no match occured */
FR_ACCOUNT      0x000400   /* count packet bytes */
FR_KEEPFRAG     0x000800   /* keep fragment information */
FR_KEEPSTATE    0x001000   /* keep 'connection' state information */
```

```
FR_INACTIVE    0x002000
FR_QUICK       0x004000  /* match & stop processing list */
FR_FASTROUTE   0x008000  /* bypass normal routing */
FR_CALLNOW     0x010000  /* call another function (fr_func) if matches */
FR_DUP         0x020000  /* duplicate the packet */
FR_LOGORBLOCK  0x040000  /* block the packet if it can't be logged */
FR_NOTSRCIP    0x080000  /* not the src IP# */
FR_NOTDSTIP    0x100000  /* not the dst IP# */
FR_AUTH        0x200000  /* use authentication */
FR_PREAUTH     0x400000  /* require preauthentication */
```

Values for fr_scomp and fr_dcomp (source and destination port value comparisons) :

```
FR_NONE       0
FR_EQUAL      1
FR_NEQUAL     2
FR_LESST      3
FR_GREATERT   4
FR_LESSTE     5
FR_GREATERTE  6
FR_OUTRANGE   7
FR_INRANGE    8
```

The third ioctl, SIOCIPFFL, flushes either the input filter list, the output filter list or both and it returns the number of filters removed from the list(s). The values which it will take and recognise are FR_INQUE and FR_OUTQUE (see above). This ioctl is also implemented for **/dev/ipstate** and will flush all state tables entries if passed 0 or just all those which are not established if passed 1.

General Logging Flags

There are two flags which can be set to log packets independantly of the rules used. These allow for packets which are either passed or blocked to be logged. To set (and clear)/get these flags, two ioctls are provided:

SIOCSETFF — Takes an unsigned integer as the parameter. The flags are then set to those provided (clearing/setting all in one).

```
FF_LOGPASS    0x10000000
FF_LOGBLOCK   0x20000000
FF_LOGNOMATCH 0x40000000
FF_BLOCKNONIP 0x80000000   /* Solaris 2.x only */
```

SIOCGETFF — Takes a pointer to an unsigned integer as the parameter. A copy of the flags currently in used is copied to user space.

Filter statistics

Statistics on the various operations performed by this package on packets is kept inside the kernel. These statistics apply to packets traversing through the kernel. To retrieve this structure, use this ioctl:

```
ioctl(fd, SIOCGETFS, struct friostat *)

struct friostat     {
    struct filterstats  f_st[2];
    struct frentry      *f_fin[2];
    struct frentry      *f_fout[2];
    struct frentry      *f_acctin[2];
    struct frentry      *f_acctout[2];
    struct frentry      *f_auth;
```

```
        u_long  f_froute[2];
        int     f_active;       /* 1 or 0 - active rule set */
        int     f_defpass;      /* default pass - from fr_pass */
        int     f_running;      /* 1 if running, else 0 */
        int     f_logging;      /* 1 if enabled, else 0 */
        char    f_version[32];  /* version string */
};

struct  filterstats {
        u_long  fr_pass;        /* packets allowed */
        u_long  fr_block;       /* packets denied */
        u_long  fr_nom;         /* packets which don't match any rule */
        u_long  fr_ppkl;        /* packets allowed and logged */
        u_long  fr_bpkl;        /* packets denied and logged */
        u_long  fr_npkl;        /* packets unmatched and logged */
        u_long  fr_pkl;         /* packets logged */
        u_long  fr_skip;        /* packets to be logged but buffer full */
        u_long  fr_ret;         /* packets for which a return is sent */
        u_long  fr_acct;        /* packets for which counting was performed */
        u_long  fr_bnfr;        /* bad attempts to allocate fragment state */
        u_long  fr_nfr;         /* new fragment state kept */
        u_long  fr_cfr;         /* add new fragment state but complete pkt */
        u_long  fr_bads;        /* bad attempts to allocate packet state */
        u_long  fr_ads;         /* new packet state kept */
        u_long  fr_chit;        /* cached hit */
        u_long  fr_pull[2];     /* good and bad pullup attempts */
#if SOLARIS
        u_long  fr_notdata;     /* PROTO/PCPROTO that have no data */
        u_long  fr_nodata;      /* mblks that have no data */
        u_long  fr_bad;         /* bad IP packets to the filter */
        u_long  fr_notip;       /* packets passed through no on ip queue */
        u_long  fr_drop;        /* packets dropped - no info for them! */
#endif
};
```

If we wanted to retrieve all the statistics and reset the counters back to 0, then the ioctl() call would be made to SIOCFRZST rather than SIOCGETFS. In addition to the statistics above, each rule keeps a hit count, counting both number of packets and bytes. To reset these counters for a rule, load the various rule information into a frentry structure and call SIOCZRLST.

Swapping Active lists

IP Filter supports two lists of rules for filtering and accounting: an active list and an inactive list. This allows for large scale rule base changes to be put in place atomically with otherwise minimal interruption. Which of the two is active can be changed using the SIOCSWAPA ioctl. It is important to note that no passed argument is recognised and that the value returned is that of the list which is now inactive.

FILES

/dev/ipauth
/dev/ipl
/dev/ipnat
/dev/ipstate

SEE ALSO

ipl(4), ipnat(4), ipf(5), ipf(8), ipfstat(8)

IPL(4)

NAME

ipl – IP packet log device

DESCRIPTION

The **ipl** pseudo device's purpose is to provide an easy way to gather packet headers of packets you wish to log. If a packet header is to be logged, the entire header is logged (including any IP options – TCP/UDP options are not included when it calculates header size) or not at all. The packet contents are also logged after the header. If the log reader is busy or otherwise unable to read log records, upto IPLLOGSIZE (8192 is the default) bytes of data are stored.

Prepending every packet header logged is a structure containing information relevant to the packet following and why it was logged. The structure's format is as follows:

```
/*
 * Log structure.  Each packet header logged is prepended by one of these.
 * Following this in the log records read from the device will be an ipflog
 * structure which is then followed by any packet data.
 */
typedef struct iplog  {
     u_long  ipl_sec;
     u_long  ipl_usec;
     u_int   ipl_len;
     u_int   ipl_count;
     size_t  ipl_dsize;
     struct  iplog  *ipl_next;
} iplog_t;


typedef struct  ipflog  {
#if (defined(NetBSD) && (NetBSD <= 1991011) && (NetBSD >= 199603))
     u_char  fl_ifname[IFNAMSIZ];
#else
     u_int   fl_unit;
     u_char  fl_ifname[4];
#endif
     u_char  fl_plen;     /* extra data after hlen */
     u_char  fl_hlen;     /* length of IP headers saved */
     u_short fl_rule;     /* assume never more than 64k rules, total */
     u_32_t  fl_flags;
} ipflog_t;
```

When reading from the **ipl** device, it is necessary to call read(2) with a buffer big enough to hold at least 1 complete log record - reading of partial log records is not supported.

If the packet contents is more then 128 bytes when **log body** is used, then only 128 bytes of the packet contents is logged.

Although it is only possible to read from the **ipl** device, opening it for writing is required when using an ioctl which changes any kernel data.

The ioctls which are loaded with this device can be found under **ipf(4)**. The ioctls which are for use with logging and don't affect the filter are:

```
ioctl(fd, SIOCIPFFB, int *)
ioctl(fd, FIONREAD, int *)
```

The SIOCIPFFB ioctl flushes the log buffer and returns the number of bytes flushed. FIONREAD returns the number of bytes currently used for storing log data. If IPFILTER_LOG is not defined when compiling, SIOCIPFFB is not available and FIONREAD will return but not do anything.

There is currently no support for non-blocking IO with this device, meaning all read operations should be considered blocking in nature (if there is no data to read, it will sleep until some is made available).

SEE ALSO

ipf(4)

BUGS

Packet headers are dropped when the internal buffer (static size) fills.

FILES

/dev/ipl

NAME

ipnat – Network Address Translation kernel interface

SYNOPSIS

```
#include <netinet/ip_compat.h>
#include <netinet/ip_fil.h>
#include <netinet/ip_proxy.h>
#include <netinet/ip_nat.h>
```

IOCTLS

To add and delete rules to the NAT list, two 'basic' ioctls are provided for use. The ioctl's are called as:

```
ioctl(fd, SIOCADNAT, struct ipnat **)
ioctl(fd, SIOCRMNAT, struct ipnat **)
ioctl(fd, SIOCGNATS, struct natstat **)
ioctl(fd, SIOCGNATL, struct natlookup **)
```

Unlike **ipf(4)**, there is only a single list supported by the kernel NAT interface. An inactive list which can be swapped to is not currently supported.

These ioctl's are implemented as being routing ioctls and thus the same rules for the various routing ioctls and the file descriptor are employed, mainly being that the fd must be that of the device associated with the module (i.e., /dev/ipl).

The strcture used with the NAT interface is described below:

```
typedef struct ipnat {
    struct ipnat *in_next;
    void *in_ifp;
    u_short in_flags;
    u_short in_pnext;
    u_short in_port[2];
    struct in_addr in_in[2];
    struct in_addr in_out[2];
    struct in_addr in_nextip;
    int in_space;
    int in_redir; /* 0 if it's a mapping, 1 if it's a hard redir */
    char in_ifname[IFNAMSIZ];
} ipnat_t;

#define in_pmin      in_port[0]    /* Also holds static redir port */
#define in_pmax      in_port[1]
#define in_nip       in_nextip.s_addr
#define in_inip      in_in[0].s_addr
#define in_inmsk     in_in[1].s_addr
#define in_outip     in_out[0].s_addr
#define in_outmsk    in_out[1].s_addr
```

Recognised values for in_redir:

```
#define NAT_MAP      0
#define NAT_REDIRECT 1
```

NAT statistics Statistics on the number of packets mapped, going in and out are kept, the number of times a new entry is added and deleted (through expiration) to the NAT table and the current usage level of the NAT table.

Pointers to the NAT table inside the kernel, as well as to the top of the internal NAT lists constructed with the **SIOCADNAT** ioctls. The table itself is a hash table of size NAT_SIZE (default size is 367).

IPNAT(4)

To retrieve the statistics, the **SIOCGNATS** ioctl must be used, with the appropriate structure passed by reference, as follows:

```
ioctl(fd, SIOCGNATS, struct natstat *)
```

```
typedef struct  natstat {
    u_long  ns_mapped[2];
    u_long  ns_added;
    u_long  ns_expire;
    u_long  ns_inuse;
    nat_t  ***ns_table;
    ipnat_t *ns_list;
} natstat_t;
```

BUGS

It would be nice if there were more flexibility when adding and deleting filter rules.

FILES

/dev/ipnat

SEE ALSO

ipf(4), ipnat(5), ipf(8), ipnat(8), ipfstat(8)

NAME

ipf, ipf.conf – IP packet filter rule syntax

DESCRIPTION

A rule file for **ipf** may have any name or even be stdin. As **ipfstat** produces parseable rules as output when displaying the internal kernel filter lists, it is quite plausible to use its output to feed back into **ipf.** Thus, to remove all filters on input packets, the following could be done:

```
# ipfstat -i | ipf -rf -
```

GRAMMAR

The format used by **ipf** for construction of filtering rules can be described using the following grammar in BNF:

```
filter-rule = [ insert ] action in-out [ options ] [ tos ] [ ttl ]
              [ proto ] [ ip ] [ group ].

insert      = "@" decnumber .
action      = block | "pass" | log | "count" | skip | auth | call .
in-out      = "in" | "out" .
options     = [ log ] [ "quick" ] [ "on" interface-name [ dup ] [ froute ] ] .
tos   = "tos" decnumber | "tos" hexnumber .
ttl   = "ttl" decnumber .
proto = "proto" protocol .
ip    = srcdst [ flags ] [ with withopt ] [ icmp ] [ keep ] .
group = [ "head" decnumber ] [ "group" decnumber ] .

block = "block" [ return-icmp[return-code] | "return-rst" ] .
auth    = "auth" | "preauth" .
log   = "log" [ "body" ] [ "first" ] [ "or-block" ] [ "level" loglevel ] .
call  = "call" [ "now" ] function-name .
skip  = "skip" decnumber .
dup   = "dup-to" interface-name[":"ipaddr] .
froute      = "fastroute" | "to" interface-name .
protocol = "tcp/udp" | "udp" | "tcp" | "icmp" | decnumber .
srcdst      = "all" | fromto .
fromto      = "from" [ "!" ] object "to" [ "!" ] object .

return-icmp = "return-icmp" | "return-icmp-as-dest" .
object      = addr [ port-comp | port-range ] .
addr  = "any" | nummask | host-name [ "mask" ipaddr | "mask" hexnumber ] .
port-comp = "port" compare port-num .
port-range = "port" port-num range port-num .
flags = "flags" flag { flag } [ "/" flag { flag } ] .
with  = "with" | "and" .
icmp  = "icmp-type" icmp-type [ "code" decnumber ] .
return-code = "("icmp-code")" .
keep  = "keep" "state" | "keep" "frags" .
loglevel = facility"."priority | priority .

nummask     = host-name [ "/" decnumber ] .
host-name = ipaddr | hostname | "any" .
ipaddr      = host-num "." host-num "." host-num "." host-num .
host-num = digit [ digit [ digit ] ] .
port-num = service-name | decnumber .
```

```
withopt = [ "not" | "no" ] opttype [ withopt ] .
opttype = "ipopts" | "short" | "frag" | "opt" ipopts  .
optname     = ipopts [ "," optname ] .
ipopts  = optlist | "sec-class" [ secname ] .
secname     = seclvl [ "," secname ] .
seclvl  = "unclass" | "confid" | "reserv-1" | "reserv-2" | "reserv-3" |
        "reserv-4" | "secret" | "topsecret" .
icmp-type = "unreach" | "echo" | "echorep" | "squench" | "redir" |
          "timex" | "paramprob" | "timest" | "timestrep" | "inforeq" |
          "inforep" | "maskreq" | "maskrep"  | decnumber .
icmp-code = decumber | "net-unr" | "host-unr" | "proto-unr" | "port-unr" |
          "needfrag" | "srcfail" | "net-unk" | "host-unk" | "isolate" |
          "net-prohib" | "host-prohib" | "net-tos" | "host-tos" |
          "filter-prohib" | "host-preced" | "cutoff-preced" .
optlist     = "nop" | "rr" | "zsu" | "mtup" | "mtur" | "encode" | "ts" |
        "tr" | "sec" | "lsrr" | "e-sec" | "cipso" | "satid" | "ssrr" |
        "addext" | "visa" | "imitd" | "eip" | "finn" .
facility = "kern" | "user" | "mail" | "daemon" | "auth" | "syslog" |
         "lpr" | "news" | "uucp" | "cron" | "ftp" | "authpriv" |
         "audit" | "logalert" | "local0" | "local1" | "local2" |
         "local3" | "local4" | "local5" | "local6" | "local7" .
priority = "emerg" | "alert" | "crit" | "err" | "warn" | "notice" |
         "info" | "debug" .

hexnumber = "0" "x" hexstring .
hexstring = hexdigit [ hexstring ] .
decnumber = digit [ decnumber ] .

compare = "=" | "!=" | "<" | ">" | "<=" | ">=" | "eq" | "ne" | "lt" |
        "gt" | "le" | "ge" .
range = "<>" | "><" .
hexdigit = digit | "a" | "b" | "c" | "d" | "e" | "f" .
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
flag  = "F" | "S" | "R" | "P" | "A" | "U" .
```

This syntax is somewhat simplified for readability, some combinations that match this grammar are disallowed by the software because they do not make sense (such as tcp **flags** for non-TCP packets).

FILTER RULES

The "briefest" valid rules are (currently) no-ops and are of the form:

block in all
pass in all
log out all
count in all

Filter rules are checked in order, with the last matching rule determining the fate of the packet (but see the **quick** option, below).

Filters are installed by default at the end of the kernel's filter lists, prepending the rule with **@n** will cause it to be inserted as the n'th entry in the current list. This is especially useful when modifying and testing active filter rulesets. See ipf(1) for more information.

ACTIONS

The action indicates what to do with the packet if it matches the rest of the filter rule. Each rule MUST have an action. The following actions are recognised:

block indicates that the packet should be flagged to be dropped. In response to blocking a packet, the filter may be instructed to send a reply packet, either an ICMP packet (**return-icmp**), an ICMP

packet masquerading as being from the original packet's destination (**return-icmp-as-dest**), or a TCP "reset" (**return-rst**). An ICMP packet may be generated in response to any IP packet, and its type may optionally be specified, but a TCP reset may only be used with a rule which is being applied to TCP packets. When using **return-icmp** or **return-icmp-as-dest**, it is possible to specify the actual unreachable 'type'. That is, whether it is a network unreachable, port unreachable or even administratively prohibitied. This is done by enclosing the ICMP code associated with it in parenthesis directly following **return-icmp** or **return-icmp-as-dest** as follows:

block return-icmp(11) ...

Would return a Type-Of-Service (TOS) ICMP unreachable error.

pass will flag the packet to be let through the filter.

log causes the packet to be logged (as described in the LOGGING section below) and has no effect on whether the packet will be allowed through the filter.

count causes the packet to be included in the accounting statistics kept by the filter, and has no effect on whether the packet will be allowed through the filter. These statistics are viewable with ipfstat(8).

call this action is used to invoke the named function in the kernel, which must conform to a specific calling interface. Customised actions and semantics can thus be implemented to supplement those available. This feature is for use by knowledgeable hackers, and is not currently documented.

skip <n>

causes the filter to skip over the next *n* filter rules. If a rule is inserted or deleted inside the region being skipped over, then the value of *n* is adjusted appropriately.

auth this allows authentication to be performed by a user-space program running and waiting for packet information to validate. The packet is held for a period of time in an internal buffer whilst it waits for the program to return to the kernel the *real* flags for whether it should be allowed through or not. Such a program might look at the source address and request some sort of authentication from the user (such as a password) before allowing the packet through or telling the kernel to drop it if from an unrecognised source.

preauth

tells the filter that for packets of this class, it should look in the pre-authenticated list for further clarification. If no further matching rule is found, the packet will be dropped (the FR_PREAUTH is not the same as FR_PASS). If a further matching rule is found, the result from that is used in its instead. This might be used in a situation where a person *logs in* to the firewall and it sets up some temporary rules defining the access for that person.

The next word must be either **in** or **out**. Each packet moving through the kernel is either inbound (just been received on an interface, and moving towards the kernel's protocol processing) or outbound (transmitted or forwarded by the stack, and on its way to an interface). There is a requirement that each filter rule explicitly state which side of the I/O it is to be used on.

OPTIONS

The list of options is brief, and all are indeed optional. Where options are used, they must be present in the order shown here. These are the currently supported options:

log indicates that, should this be the last matching rule, the packet header will be written to the **ipl** log (as described in the LOGGING section below).

quick allows "short-cut" rules in order to speed up the filter or override later rules. If a packet matches a filter rule which is marked as **quick**, this rule will be the last rule checked, allowing a "short-circuit" path to avoid processing later rules for this packet. The current status of the packet (after any effects of the current rule) will determine whether it is passed or blocked.

If this option is missing, the rule is taken to be a "fall-through" rule, meaning that the result of the match (block/pass) is saved and that processing will continue to see if there are any more matches.

on allows an interface name to be incorporated into the matching procedure. Interface names are as printed by "netstat –i". If this option is used, the rule will only match if the packet is going through

that interface in the specified direction (in/out). If this option is absent, the rule is taken to be applied to a packet regardless of the interface it is present on (i.e. on all interfaces). Filter rulesets are common to all interfaces, rather than having a filter list for each interface.

This option is especially useful for simple IP-spoofing protection: packets should only be allowed to pass inbound on the interface from which the specified source address would be expected, others may be logged and/or dropped.

dup-to causes the packet to be copied, and the duplicate packet to be sent outbound on the specified interface, optionally with the destination IP address changed to that specified. This is useful for off-host logging, using a network sniffer.

to causes the packet to be moved to the outbound queue on the specified interface. This can be used to circumvent kernel routing decisions, and even to bypass the rest of the kernel processing of the packet (if applied to an inbound rule). It is thus possible to construct a firewall that behaves transparently, like a filtering hub or switch, rather than a router. The **fastroute** keyword is a synonym for this option.

MATCHING PARAMETERS

The keywords described in this section are used to describe attributes of the packet to be used when determining whether rules match or don't match. The following general-purpose attributes are provided for matching, and must be used in this order:

tos packets with different Type-Of-Service values can be filtered. Individual service levels or combinations can be filtered upon. The value for the TOS mask can either be represented as a hex number or a decimal integer value.

ttl packets may also be selected by their Time-To-Live value. The value given in the filter rule must exactly match that in the packet for a match to occur. This value can only be given as a decimal integer value.

proto allows a specific protocol to be matched against. All protocol names found in **/etc/protocols** are recognised and may be used. However, the protocol may also be given as a DECIMAL number, allowing for rules to match your own protocols, or new ones which would out-date any attempted listing.

The special protocol keyword **tcp/udp** may be used to match either a TCP or a UDP packet, and has been added as a convenience to save duplication of otherwise-identical rules.

The **from** and **to** keywords are used to match against IP addresses (and optionally port numbers). Rules must specify BOTH source and destination parameters.

IP addresses may be specified in one of two ways: as a numerical address/mask, or as a hostname **mask** netmask. The hostname may either be a valid hostname, from either the hosts file or DNS (depending on your configuration and library) or of the dotted numeric form. There is no special designation for networks but network names are recognised. Note that having your filter rules depend on DNS results can introduce an avenue of attack, and is discouraged.

There is a special case for the hostname **any** which is taken to be 0.0.0.0/0 (see below for mask syntax) and matches all IP addresses. Only the presence of "any" has an implied mask, in all other situations, a hostname MUST be accompanied by a mask. It is possible to give "any" a hostmask, but in the context of this language, it is non-sensical.

The numerical format "x/y" indicates that a mask of y consecutive 1 bits set is generated, starting with the MSB, so a y value of 16 would give 0xffff0000. The symbolic "x **mask** y" indicates that the mask y is in dotted IP notation or a hexadecimal number of the form 0x12345678. Note that all the bits of the IP address indicated by the bitmask must match the address on the packet exactly; there isn't currently a way to invert the sense of the match, or to match ranges of IP addresses which do not express themselves easily as bitmasks (anthropomorphization; it's not just for breakfast anymore).

If a **port** match is included, for either or both of source and destination, then it is only applied to TCP and UDP packets. If there is no **proto** match parameter, packets from both protocols are compared. This is

equivalent to "proto tcp/udp". When composing **port** comparisons, either the service name or an integer port number may be used. Port comparisons may be done in a number of forms, with a number of comparison operators, or port ranges may be specified. When the port appears as part of the **from** object, it matches the source port number, when it appears as part of the **to** object, it matches the destination port number. See the examples for more information.

The **all** keyword is essentially a synonym for "from any to any" with no other match parameters.

Following the source and destination matching parameters, the following additional parameters may be used:

with is used to match irregular attributes that some packets may have associated with them. To match the presence of IP options in general, use **with ipopts**. To match packets that are too short to contain a complete header, use **with short**. To match fragmented packets, use **with frag**. For more specific filtering on IP options, individual options can be listed.

Before any parameter used after the **with** keyword, the word **not** or **no** may be inserted to cause the filter rule to only match if the option(s) is not present.

Multiple consecutive **with** clauses are allowed. Alternatively, the keyword **and** may be used in place of **with**, this is provided purely to make the rules more readable ("with ... and ..."). When multiple clauses are listed, all those must match to cause a match of the rule.

flags is only effective for TCP filtering. Each of the letters possible represents one of the possible flags that can be set in the TCP header. The association is as follows:

F - FIN
S - SYN
R - RST
P - PUSH
A - ACK
U - URG

The various flag symbols may be used in combination, so that "SA" would represent a SYN-ACK combination present in a packet. There is nothing preventing the specification of combinations, such as "SFR", that would not normally be generated by law-abiding TCP implementations. However, to guard against weird aberrations, it is necessary to state which flags you are filtering against. To allow this, it is possible to set a mask indicating which TCP flags you wish to compare (i.e., those you deem significant). This is done by appending "/<flags>" to the set of TCP flags you wish to match against, e.g.:

```
... flags S
                # becomes "flags S/AUPRFS" and will match
                # packets with ONLY the SYN flag set.

... flags SA
                # becomes "flags SA/AUPRFSC" and will match any
                # packet with only the SYN and ACK flags set.

... flags S/SA
                # will match any packet with just the SYN flag set
                # out of the SYN-ACK pair; the common "establish"
                # keyword action. "S/SA" will NOT match a packet
                # with BOTH SYN and ACK set, but WILL match "SFP".
```

icmp-type
is only effective when used with **proto icmp** and must NOT be used in conjuction with **flags**. There are a number of types, which can be referred to by an abbreviation recognised by this language, or the numbers with which they are associated can be used. The most important from a security point of view is the ICMP redirect.

KEEP HISTORY

The second last parameter which can be set for a filter rule is whether or not to record historical information for that packet, and what sort to keep. The following information can be kept:

state keeps information about the flow of a communication session. State can be kept for TCP, UDP, and ICMP packets.

frags keeps information on fragmented packets, to be applied to later fragments.

allowing packets which match these to flow straight through, rather than going through the access control list.

GROUPS

The last pair of parameters control filter rule "grouping". By default, all filter rules are placed in group 0 if no other group is specified. To add a rule to a non-default group, the group must first be started by creating a group *head*. If a packet matches a rule which is the *head* of a group, the filter processing then switches to the group, using that rule as the default for the group. If **quick** is used with a **head** rule, rule processing isn't stopped until it has returned from processing the group.

A rule may be both the head for a new group and a member of a non-default group (**head** and **group** may be used together in a rule).

head <n>

indicates that a new group (number n) should be created.

group <n>

indicates that the rule should be put in group (number n) rather than group 0.

LOGGING

When a packet is logged, with either the **log** action or option, the headers of the packet are written to the **ipl** packet logging psuedo-device. Immediately following the **log** keyword, the following qualifiers may be used (in order):

body indicates that the first 128 bytes of the packet contents will be logged after the headers.

first If log is being used in conjunction with a "keep" option, it is recommended that this option is also applied so that only the triggering packet is logged and not every packet which thereafter matches state information.

or-block

indicates that, if for some reason the filter is unable to log the packet (such as the log reader being too slow) then the rule should be interpreted as if the action was **block** for this packet.

level <loglevel>

indicates what logging facility and priority, or just priority with the default facility being used, will be used to log information about this packet using ipmon's -s option.

See ipl(4) for the format of records written to this device. The ipmon(8) program can be used to read and format this log.

EXAMPLES

The **quick** option is good for rules such as:

```
block in quick from any to any with ipopts
```

which will match any packet with a non-standard header length (IP options present) and abort further processing of later rules, recording a match and also that the packet should be blocked.

The "fall-through" rule parsing allows for effects such as this:

```
block in from any to any port < 6000
pass in from any to any port >= 6000
block in from any to any port > 6003
```

which sets up the range 6000-6003 as being permitted and all others being denied. Note that the effect of

the first rule is overridden by subsequent rules. Another (easier) way to do the same is:

```
block in from any to any port 6000 <> 6003
pass in from any to any port 5999 >< 6004
```

Note that both the "block" and "pass" are needed here to effect a result as a failed match on the "block" action does not imply a pass, only that the rule hasn't taken effect. To then allow ports < 1024, a rule such as:

```
pass in quick from any to any port < 1024
```

would be needed before the first block. To create a new group for processing all inbound packets on le0/le1/lo0, with the default being to block all inbound packets, we would do something like:

```
block in all
block in quick on le0 all head 100
block in quick on le1 all head 200
block in quick on lo0 all head 300
```

and to then allow ICMP packets in on le0, only, we would do:

```
pass in proto icmp all group 100
```

Note that because only inbound packets on le0 are used processed by group 100, there is no need to respecify the interface name. Likewise, we could further breakup processing of TCP, etc, as follows:

```
block in proto tcp all head 110 group 100
pass in from any to any port = 23 group 110
```

and so on. The last line, if written without the groups would be:

```
pass in on le0 proto tcp from any to any port = telnet
```

Note, that if we wanted to say "port = telnet", "proto tcp" would need to be specified as the parser interprets each rule on its own and qualifies all service/port names with the protocol specified.

FILES

/dev/ipauth
/dev/ipl
/dev/ipstate
/etc/hosts
/etc/services

SEE ALSO

ipftest(1), iptest(1), mkfilters(1), ipf(4), ipnat(5), ipf(8), ipfstat(8)

NAME

ipnat, ipnat.conf – IP NAT file format

DESCRIPTION

The format for files accepted by ipnat is described by the following grammar:

```
ipmap :: = mapblock | redir | map .

map ::= mapit ifname ipmask "->" ipmask [ mapport ] .
map ::= mapit ifname fromto "->" ipmask [ mapport ] .
mapblock ::= "map-block" ifname ipmask "->" ipmask [ ports ] .
redir ::= "rdr" ifname ipmask dport "->" ip [ "," ip ] [ ports ] options .

dport ::= "port" portnum [ "-" portnum ] .
ports ::= "ports" numports | "auto" .
mapit ::= "map" | "bimap" .
fromto ::= "from" object "to" object .
ipmask ::= ip "/" bits | ip "/" mask | ip "netmask" mask .
mapport ::= "portmap" tcpudp portnumber ":" portnumber .
options ::= [ tcpudp ] [ rr ] .

object  = addr [ port-comp | port-range ] .
addr    = "any" | nummask | host-name [ "mask" ipaddr | "mask" hexnumber ] .
port-comp = "port" compare port-num .
port-range = "port" port-num range port-num .

rr ::= "round-robin" .
tcpudp ::= "tcp" | "udp" | "tcp/udp" .
portnumber ::= number { numbers } | "auto" .
ifname ::= 'A' - 'Z' { 'A' - 'Z' } numbers .

numbers ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
```

For standard NAT functionality, a rule should start with **map** and then proceeds to specify the interface for which outgoing packets will have their source address rewritten.

Packets which will be rewritten can only be selected by matching the original source address. A netmask must be specified with the IP address.

The address selected for replacing the original is chosen from an IP#/netmask pair. A netmask of all 1's indicating a hostname is valid. A netmask of 31 1's (255.255.255.254) is considered invalid as there is no space for allocating host IP#'s after consideration for broadcast and network addresses.

When remapping TCP and UDP packets, it is also possible to change the source port number. Either TCP or UDP or both can be selected by each rule, with a range of port numbers to remap into given as **port-number:port-number**.

COMMANDS

There are four commands recognised by IP Filter's NAT code:

map that is used for mapping one address or network to another in an unregulated round robin fashion;

rdr that is used for redirecting packets to one IP address and port pair to another;

bimap for setting up bidirectional NAT between an external IP address and an internal IP address and

map-block
which sets up static IP address based translation, based on a algorithm to squeeze the addresses to be translated into the destination range.

MATCHING

For basic NAT and redirection of packets, the address subject to change is used along with its protocol to check if a packet should be altered. The packet *matching* part of the rule is to the left of the "->" in each rule.

Matching of packets has now been extended to allow more complex compares. In place of the address which is to be translated, an IP address and port number comparison can be made using the same expressions available with **ipf**. A simple NAT rule could be written as:

map de0 10.1.0.0/16 -> 201.2.3.4/32

or as

map de0 from 10.1.0.0/16 to any -> 201.2.3.4/32

Only IP address and port numbers can be compared against. This is available with all NAT rules.

TRANSLATION

To the right of the "->" is the address and port specificaton which will be written into the packet providing it has already successful matched the prior constraints. The case of redirections (**rdr**) is the simpliest: the new destination address is that specified in the rule. For **map** rules, the destination address will be one for which the tuple combining the new source and destination is known to be unique. If the packet is either a TCP or UDP packet, the destination and source ports come into the equation too. If the tuple already exists, IP Filter will increment the port number first, within the available range specified with **portmap** and if there exists no unique tuple, the source address will be incremented within the specified netmask. If a unique tuple cannot be determined, then the packet will not be translated. The **map-block** is more limited in how it searches for a new, free and unique tuple, in that it will used an algorithm to determine what the new source address should be, along with the range of available ports - the IP address is never changed and nor does the port number ever exceed its alloted range.

KERNEL PROXIES

IP Filter comes with a few, simple, proxies built into the code that is loaded into the kernel to allow secondary channels to be opened without forcing the packets through a user program.

TRNSPARENT PROXIES

True transparent proxying should be performed using the redirect (**rdr**) rules directing ports to localhost (127.0.0.1) with the proxy program doing a lookup through **/dev/ipnat** to determine the real source and address of the connection.

LOAD-BALANCING

Two options for use with **rdr** are available to support primitive, *round-robin* based load balancing. The first option allows for a **rdr** to specify a second destination, as follows:

rdr le0 203.1.2.3/32 port 80 -> 203.1.2.3,203.1.2.4 port 80 tcp

This would send alternate connections to either 203.1.2.3 or 203.1.2.4. In scenarios where the load is being spread amongst a larger set of servers, you can use:

rdr le0 203.1.2.3/32 port 80 -> 203.1.2.3,203.1.2.4 port 80 tcp round-robin
rdr le0 203.1.2.3/32 port 80 -> 203.1.2.5 port 80 tcp round-robin

In this case, a connection will be redirected to 203.1.2.3, then 203.1.2.4 and then 203.1.2.5 before going back to 203.1.2.3. In accomplishing this, the rule is removed from the top of the list and added to the end, automatically, as required. This will not effect the display of rules using "ipnat -l", only the internal application order.

EXAMPLES

This section deals with the **map** command and it's variations.

To change IP#'s used internally from network 10 into an ISP provided 8 bit subnet at 209.1.2.0 through the ppp0 interface, the following would be used:

map ppp0 10.0.0.0/8 -> 209.1.2.0/24

The obvious problem here is we're trying to squeeze over 16,000,000 IP addresses into a 254 address space.

IPNAT(5)

To increase the scope, remapping for TCP and/or UDP, port remapping can be used;

map ppp0 10.0.0.0/8 -> 209.1.2.0/24 portmap tcp/udp 1025:65000

which falls only 527,566 'addresses' short of the space available in network 10. If we were to combine these rules, they would need to be specified as follows:

map ppp0 10.0.0.0/8 -> 209.1.2.0/24 portmap tcp/udp 1025:65000
map ppp0 10.0.0.0/8 -> 209.1.2.0/24

so that all TCP/UDP packets were port mapped and only other protocols, such as ICMP, only have their IP# changed. In some instaces, it is more appropriate to use the keyword **auto** in place of an actual range of port numbers if you want to guarantee simultaneous access to all within the given range. However, in the above case, it would default to 1 port per IP address, since we need to squeeze 24 bits of address space into 8. A good example of how this is used might be:

map ppp0 172.192.0.0/16 -> 209.1.2.0/24 portmap tcp/udp auto

which would result in each IP address being given a small range of ports to use (252). The problem here is that the **map** directive tells the NAT code to use the next address/port pair available for an outgoing connection, resulting in no easily discernable relation between external addresses/ports and internal ones. This is overcome by using **map-block** as follows:

map-block ppp0 172.192.0.0/16 -> 209.1.2.0/24 ports auto

For example, this would result in 172.192.0.0/24 being mapped to 209.1.2.0/32 with each address, from 172.192.0.0 to 172.192.0.255 having 252 ports of its own. As opposed to the above use of **map**, if for some reason the user of (say) 172.192.0.2 wanted 260 simultaneous connections going out, they would be limited to 252 with **map-block** but would just *move on* to the next IP address with the **map** command.

FILES

/dev/ipnat
/etc/services
/etc/hosts

SEE ALSO

ipnat(4), hosts(5), ipf(5), services(5), ipf(8), ipnat(8)

NAME

ipf – alters packet filtering lists for IP packet input and output

SYNOPSIS

ipf [**–6AdDEInoPrsUvVyzZ**] [**–l** <block|pass|nomatch>] [**–F** <i|o|a|s|S>] **–f** <*filename*> [**–f** <*filename*> [...]]

DESCRIPTION

ipf opens the filenames listed (treating "–" as stdin) and parses the file for a set of rules which are to be added or removed from the packet filter rule set.

Each rule processed by **ipf** is added to the kernel's internal lists if there are no parsing problems. Rules are added to the end of the internal lists, matching the order in which they appear when given to **ipf**.

OPTIONS

–6 This option is required to parse IPv6 rules and to have them loaded.

–A Set the list to make changes to the active list (default).

–d Turn debug mode on. Causes a hexdump of filter rules to be generated as it processes each one.

–D Disable the filter (if enabled). Not effective for loadable kernel versions.

–E Enable the filter (if disabled). Not effective for loadable kernel versions.

–F <i|o|a>

This option specifies which filter list to flush. The parameter should either be "i" (input), "o" (output) or "a" (remove all filter rules). Either a single letter or an entire word starting with the appropriate letter maybe used. This option maybe before, or after, any other with the order on the command line being that used to execute options.

–F <s|S>

To flush entries from the state table, the **-F** option is used in conjuction with either "s" (removes state information about any non-fully established connections) or "S" (deletes the entire state table). Only one of the two options may be given. A fully established connection will show up in **ipfstat -s** output as 4/4, with deviations either way indicating it is not fully established any more.

–f <filename>

This option specifies which files **ipf** should use to get input from for modifying the packet filter rule lists.

–I Set the list to make changes to the inactive list.

–l <pass|block|nomatch>

Use of the **-l** flag toggles default logging of packets. Valid arguments to this option are **pass**, **block** and **nomatch**. When an option is set, any packet which exits filtering and matches the set category is logged. This is most useful for causing all packets which don't match any of the loaded rules to be logged.

–n This flag (no-change) prevents **ipf** from actually making any ioctl calls or doing anything which would alter the currently running kernel.

–o Force rules by default to be added/deleted to/from the output list, rather than the (default) input list.

–P Add rules as temporary entries in the authentication rule table.

–r Remove matching filter rules rather than add them to the internal lists

–s Swap the active filter list in use to be the "other" one.

–U (SOLARIS 2 ONLY) Block packets travelling along the data stream which aren't recognised as IP packets. They will be printed out on the console.

–v Turn verbose mode on. Displays information relating to rule processing.

-V Show version information. This will display the version information compiled into the ipf binary and retrieve it from the kernel code (if running/present). If it is present in the kernel, information about its current state will be displayed (whether logging is active, default filtering, etc).

-y Manually resync the in-kernel interface list maintained by IP Filter with the current interface status list.

-z For each rule in the input file, reset the statistics for it to zero and display the statistics prior to them being zero'd.

-Z Zero global statistics held in the kernel for filtering only (this doesn't affect fragment or state statistics).

FILES

/dev/ipauth
/dev/ipl
/dev/ipstate

SEE ALSO

ipftest(1), mkfilters(1), ipf(4), ipl(4), ipf(5), ipfstat(8), ipmon(8), ipnat(8)

DIAGNOSTICS

Needs to be run as root for the packet filtering lists to actually be affected inside the kernel.

BUGS

If you find any, please send email to me at darrenr@pobox.com

NAME

ipfs – saves and restores information for NAT and state tables.

SYNOPSIS

ipfs [-nv] -l

ipfs [-nv] -u

ipfs [-nv] [**–d** <*dirname*>] -R

ipfs [-nv] [**–d** <*dirname*>] -W

ipfs [-nNSv] [**–f** <*filename*>] -r

ipfs [-nNSv] [**–f** <*filename*>] -w

ipfs [-nNSv] **–f** <*filename*> **–i** <if1>,<if2>

DESCRIPTION

ipfs allows state information created for NAT entries and rules using *keep state* to be locked (modification prevented) and then saved to disk, allowing for the system to experience a reboot, followed by the restoration of that information, resulting in connections not being interrupted.

OPTIONS

–d Change the default directory used with **–R** and **–W** options for saving state information.

–n Don't actually take any action that would effect information stored in the kernel or on disk.

–v Provides a verbose description of what's being done.

–i <ifname1>,<ifname2>

Change all instances of interface name ifname1 in the state save file to ifname2. Useful if you're restoring state information after a hardware reconfiguration or change.

–N Operate on NAT information.

–S Operate on filtering state information.

–u Unlock state tables in the kernel.

–l Lock state tables in the kernel.

–r Read information in from the specified file and load it into the kernel. This requires the state tables to have already been locked and does not change the lock once comlete.

–w Write information out to the specified file and from the kernel. This requires the state tables to have already been locked and does not change the lock once comlete.

–R Restores all saved state information, if any, from two files, *ipstate.ipf* and *ipnat.ipf*, stored in the */var/db/ipf* directory unless otherwise specified the **–d** option is used. The state tables are locked at the beginning of this operation and unlocked once complete.

–W Saves in-kernel state information, if any, out to two files, *ipstate.ipf* and *ipnat.ipf*, stored in the */var/db/ipf* directory unless otherwise specified the **–d** option is used. The state tables are locked at the beginning of this operation and unlocked once complete.

FILES

/var/db/ipf/ipstate.ipf
/var/db/ipf/ipnat.ipf
/dev/ipl
/dev/ipstate
/dev/ipnat

SEE ALSO

ipf(8), ipl(4), ipmon(8), ipnat(8)

DIAGNOSTICS

Perhaps the -W and -R operations should set the locking but rather than undo it, restore it to what it was previously. Fragment table information is currently not saved.

BUGS

If you find any, please send email to me at darrenr@pobox.com

NAME

ipfstat – reports on packet filter statistics and filter list

SYNOPSIS

ipfstat [**–6aAfghIinosv**] [**–d** <device>]

ipfstat -t [**–C**] [**–D** <addrport>] [**–P** <protocol>] [**–S** <addrport>] [**–T** <refresh time>] [**–d** <device>]

DESCRIPTION

ipfstat examines /dev/kmem using the symbols **_fr_flags**, **_frstats**, **_filterin**, and **_filterout**. To run and work, it needs to be able to read both /dev/kmem and the kernel itself. The kernel name defaults to **/vmunix**.

The default behaviour of **ipfstat** is to retrieve and display the accumulated statistics which have been accumulated over time as the kernel has put packets through the filter.

OPTIONS

–6 Display filter lists for IPv6, if available.

–a Display the accounting filter list and show bytes counted against each rule.

–A Display packet authentication statistics.

–C This option is only valid in combination with **–t**. Display "closed" states as well in the top. Normally, a TCP connection is not displayed when it reaches the CLOSE_WAIT protocol state. With this option enabled, all state entries are displayed.

–d <device>
Use a device other than **/dev/ipl** for interfacing with the kernel.

–D <addrport>
This option is only valid in combination with **–t**. Limit the state top display to show only state entries whose destination IP address and port match the addport argument. The addrport specification is of the form ipaddress[,port]. The ipaddress and port should be either numerical or the string "any" (specifying any ip address resp. any port). If the **–D** option is not specified, it defaults to "**–D** any,any".

–f Show fragment state information (statistics) and held state information (in the kernel) if any is present.

–g Show groups currently configured (both active and inactive).

–h Show per-rule the number of times each one scores a "hit". For use in combination with **–i**.

–i Display the filter list used for the input side of the kernel IP processing.

–I Swap between retrieving "inactive"/"active" filter list details. For use in combination with **–i**.

–n Show the "rule number" for each rule as it is printed.

–o Display the filter list used for the output side of the kernel IP processing.

–P <protocol>
This option is only valid in combination with **–t**. Limit the state top display to show only state entries that match a specific protocol. The argument can be a protocol name (as defined in **/etc/protocols**) or a protocol number. If this option is not specified, state entries for any protocol are specified.

–s Show packet/flow state information (statistics only).

–sl Show held state information (in the kernel) if any is present (no statistics).

–S <addrport>
This option is only valid in combination with **–t**. Limit the state top display to show only state entries whose source IP address and port match the addport argument. The addrport specification

is of the form ipaddress[,port]. The ipaddress and port should be either numerical or the string "any" (specifying any ip address resp. any port). If the **-S** option is not specified, it defaults to "**-S** any,any".

-t Show the state table in a way similar to they way **top(1)** shows the process table. States can be sorted using a number of different ways. This options requires **ncurses(3)** and needs to be compiled in. It may not be available on all operating systems. See below, for more information on the keys that can be used while ipfstat is in top mode.

-T <refreshtime>
This option is only valid in combination with **-t**. Specifies how often the state top display should be updated. The refresh time is the number of seconds between an update. Any postive integer can be used. The default (and minimal update time) is 1.

-v Turn verbose mode on. Displays more debugging information.

SYNOPSIS

The role of **ipfstat** is to display current kernel statistics gathered as a result of applying the filters in place (if any) to packets going in and out of the kernel. This is the default operation when no command line parameters are present.

When supplied with either **-i** or **-o**, it will retrieve and display the appropriate list of filter rules currently installed and in use by the kernel.

STATE TOP

Using the **-t** option **ipfstat** will enter the state top mode. In this mode the state table is displayed similar to the way **top** displays the process table. The **-C**, **-D**, **-P**, **-S** and **-T** commandline options can be used to restrict the state entries that will be shown and to specify the frequency of display updates.

In state top mode, the following keys can be used to influence the displayed information:

d select information to display.

l redraw the screen.

q quit the program.

s switch between different sorting criterion.

r reverse the sorting criterion.

States can be sorted by protocol number, by number of IP packets, by number of bytes and by time-to-live of the state entry. The default is to sort by the number of bytes. States are sorted in descending order, but you can use the **r** key to sort them in ascending order.

STATE TOP LIMITATIONS

It is currently not possible to interactively change the source, destination and protocol filters or the refreh frequency. This must be done from the command line.

The screen must have at least 80 columns. This is however not checked.

Only the first X-5 entries that match the sort and filter criteria are displayed (where X is the number of rows on the display. There is no way to see more entries.

No support for IPv6

FILES

/dev/kmem
/dev/ipl
/dev/ipstate
/vmunix

SEE ALSO

ipf(8)

BUGS

none known.

NAME

ipmon – monitors /dev/ipl for logged packets

SYNOPSIS

ipmon [**–aDFhnpstvxX**] [**–N <device>**] [**–o [NSI]**] [**–O [NSI]**] [**–P <pidfile>**] [**–S <device>**] [**–f <device>**] [**<filename>**]

DESCRIPTION

ipmon opens **/dev/ipl** for reading and awaits data to be saved from the packet filter. The binary data read from the device is reprinted in human readable for, however, IP#'s are not mapped back to hostnames, nor are ports mapped back to service names. The output goes to standard output by default or a filename, if given on the command line. Should the –s option be used, output is instead sent to **syslogd(8)**. Messages sent via syslog have the day, month and year removed from the message, but the time (including microseconds), as recorded in the log, is still included.

Messages generated by ipmon consist of whitespace separated fields. Fields common to all messages are:

1. The date of packet receipt. This is suppressed when the message is sent to syslog.

2. The time of packet receipt. This is in the form HH:MM:SS.F, for hours, minutes seconds, and fractions of a second (which can be several digits long).

3. The name of the interface the packet was processed on, e.g., **we1**.

4. The group and rule number of the rule, e.g., **@0:17**. These can be viewed with **ipfstat -n**.

5. The action: **p** for passed or **b** for blocked.

6. The addresses. This is actually three fields: the source address and port (separted by a comma), the -> symbol, and the destination address and port. E.g.: **209.53.17.22,80 -> 198.73.220.17,1722**.

7. **PR** followed by the protocol name or number, e.g., **PR tcp**.

8. **len** followed by the header length and total length of the packet, e.g., **len 20 40**.

If the packet is a TCP packet, there will be an additional field starting with a hyphen followed by letters corresponding to any flags that were set. See the ipf.conf manual page for a list of letters and their flags.

If the packet is an ICMP packet, there will be two fields at the end, the first always being 'icmp', and the next being the ICMP message and submessage type, separated by a slash, e.g., **icmp 3/3** for a port unreachable message.

In order for **ipmon** to properly work, the kernel option **IPFILTER_LOG** must be turned on in your kernel. Please see **options(4)** for more details.

OPTIONS

–a Open all of the device logfiles for reading log entries from. All entries are displayed to the same output 'device' (stderr or syslog).

–D Cause ipmon to turn itself into a daemon. Using subshells or backgrounding of ipmon is not required to turn it into an orphan so it can run indefinately.

–f <device>
specify an alternative device/file from which to read the log information for normal IP Filter log records.

–F Flush the current packet log buffer. The number of bytes flushed is displayed, even should the result be zero.

–n IP addresses and port numbers will be mapped, where possible, back into hostnames and service names.

–N <device>
Set the logfile to be opened for reading NAT log records from to <device>.

–o Specify which log files to actually read data from. N - NAT logfile, S - State logfile, I - normal IP Filter logfile. The **-a** option is equivalent to using **-o NSI**.

-O Specify which log files you do not wish to read from. This is most sensibly used with the **-a**. Letters available as paramters to this are the same as for **-o**.

-p Cause the port number in log messages to always be printed as a number and never attempt to look it up as from */etc/services*, etc.

-P <pidfile>
Write the pid of the ipmon process to a file. By default this is */*/*etc*/*opt*/*ipf*/*ipmon.pid* (Solaris), */var*/*run*/*ipmon.pid* (44BSD or later) or */etc*/*ipmon.pid* for all others.

-s Packet information read in will be sent through syslogd rather than saved to a file. The default facility when compiled and installed is **local0**. The following levels are used:

LOG_INFO – packets logged using the "log" keyword as the action rather than pass or block.

LOG_NOTICE – packets logged which are also passed

LOG_WARNING – packets logged which are also blocked

LOG_ERR – packets which have been logged and which can be considered "short".

-S <device>
Set the logfile to be opened for reading state log records from to <device>.

-t read the input file/device in a manner akin to tail(1).

-v show tcp window, ack and sequence fields.

-x show the packet data in hex.

-X show the log header record data in hex.

DIAGNOSTICS

ipmon expects data that it reads to be consistent with how it should be saved and will abort if it fails an assertion which detects an anomaly in the recorded data.

FILES

/dev/ipl
/dev/ipnat
/dev/ipstate
/etc/services

SEE ALSO

ipl(4), ipf(8), ipfstat(8), ipnat(8)

BUGS